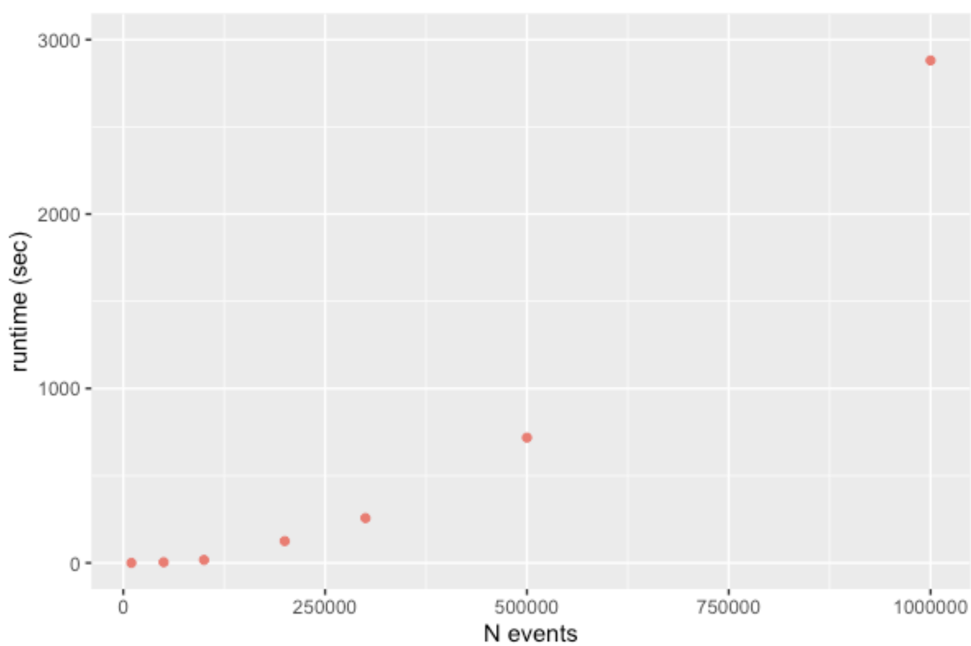
Jamie Goodin
12/17/22
SST Challenge Algorithm Rationale

Starting off (Method 1)

When first approaching this problem, I just wanted to get it sorting accurately, and sorted the events using nested `for` loops. For every event, it iterated through each existing level until it found a match (or not). But as the number of levels grew, performance spiraled, and by 250,000 inputs, it was clearly too slow. To start off, I was sorting with `O(n^2)` complexity.



Method 1 performance

Interval tree

I then wondered if there was in answer in tracking each event's neighbors and leveraging a different data structure. I researched such approaches, and found the "interval tree" data structure. When applied appropriately, it can produce an algorithmic complexity of `O(n*log(n))`.

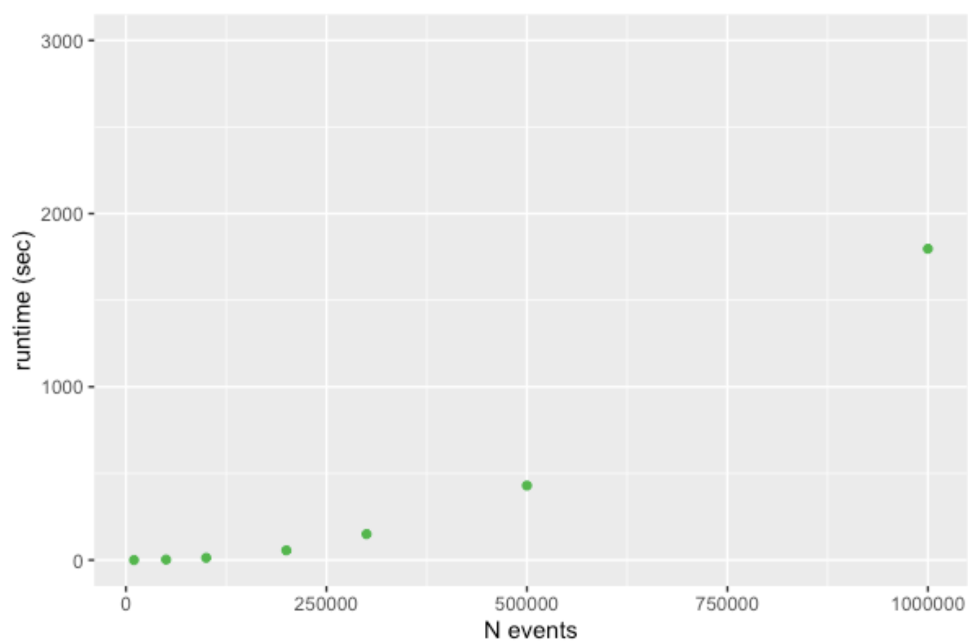
Rather than start from scratch, I tried out an Interval Tree node package. While it did sort events correctly, it was unfortunately much slower than my first solution—so much that I did not wait for it to complete larger jobs. Perhaps there was more complexity in the package's search and insert functions than I realized, or I had implemented it imperfectly. Regardless, I was unable to achieve a more efficient result using an interval tree.

Recursion and binary search

Recursion's best use cases are not performance-oriented, but I wanted to see I could use it to make searching more efficient. My implementation was functional, but only up to about 100,000 events. After that, I encountered memory issues and call stack limitations. I also tried a binary search that utilized recursion, but encountered the same issues.

A level-forward approach (Method 2)

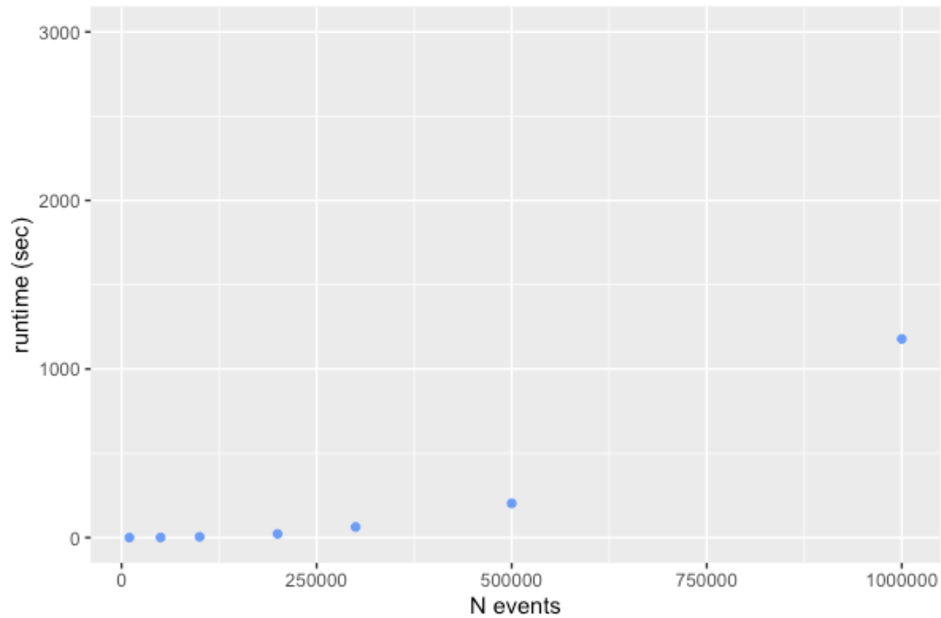
The recursive method introduced an important shift in my approach. Rather than iterating through each event and finding a level for it in a growing pool of levels, the algorithm instead filled a base level with as many events as possible. It then created and fills new levels until each event is sorted. I refactored the method to a `while` loop, and though it still had a complexity of `O(n^2)`, it was by far the fastest solution yet.



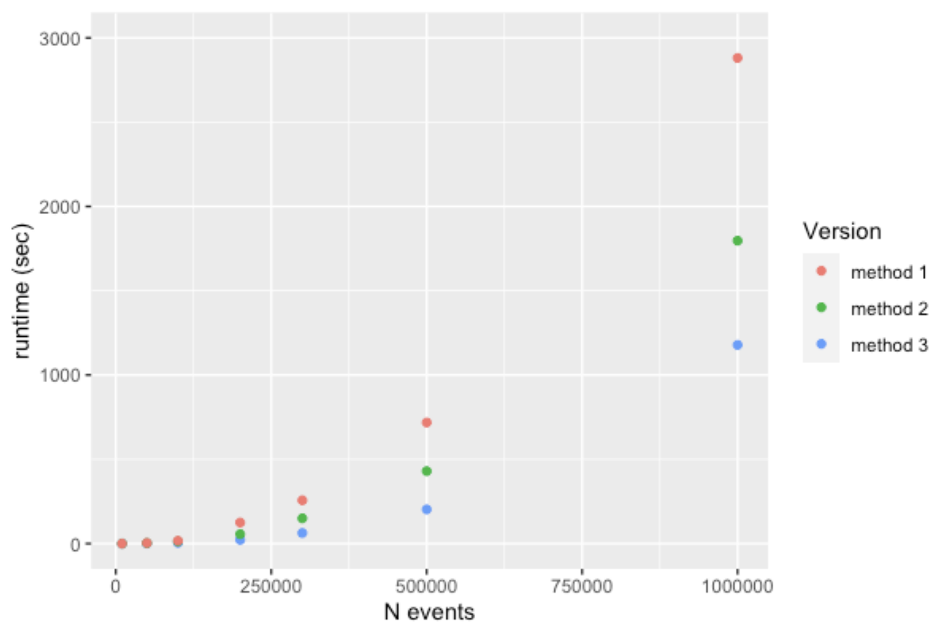
Method 2 performance

Sorted approach (Method 3)

The final and fastest method improves on the last one by taking a more intentional approach to packing the levels, an as of yet untapped source of improvement. It first sorts the inputs by `endTime`. It then compares the first event in the list's `endTime` to the next event's `startTime`, pushing the next event to the current level if it starts before the first event ends. This resulted in significantly better scaling, and a result I could be happy with.



Method 3 performance



All three methods' performance

Conclusion

After a lot of work and research, I have produced an algorithm that sorts correctly, and much more efficiently than some other options. This was a fun and interesting technical problem to analyze, and it's easy to see how it's relevant to Surgical Safety Technology's products. I look forward to discussing this challenge with your team, and learning more ways to approach it.